

iEMSs 2016 Conference

Environmental modelling and software
for supporting a sustainable future

Draft



Proceedings | Volume 4 | Pages 803-1274
**8th International Congress on Environmental
Modeling and Software (iEMSs)**
July 10-14, 2016
Toulouse, France

**Proceedings of the 8th International Congress on Environmental
Modelling and Software (iEMSs)
July 10-14, 2016, Toulouse, FRANCE.**

How to cite the full proceedings:

Sauvage, S., Sánchez-Pérez, J.M., Rizzoli, A.E. (Eds.), 2016. Proceedings of the 8th International Congress on Environmental Modelling and Software, July 10-14, Toulouse, FRANCE. ISBN: 978-88-9035-745-9

How to cite an individual paper:

Author, A., Author, B., Author, C..., 2016. This is the title of your paper. In: Sauvage, S., Sánchez-Pérez, J.M., Rizzoli, A.E. (Eds.), 2016. Proceedings of the 8th International Congress on Environmental Modelling and Software, July 10-14, Toulouse, FRANCE. ISBN: 978-88-9035-745-9

Peer Review:

Each paper has been peer reviewed by at least two independent reviewers with possible outcomes of reject, revise, and accept.

Facilitating the Design of ABM and the Code Generation to Promote Participatory Modelling

Peter Uhnak^a and Pierre Bommel^b

a. Department of Software Engineering, Faculty of Information Technology, Czech Technical University in Prague, Czech Republic ; **b.** CIRAD, Green Research Unit, Montpellier, France & University of Costa Rica, San José, Costa Rica

Abstract:

Designing and implementing an ABM is a hard task that needs programming skills besides design skills. This is especially true when dealing with Participatory Modelling (PM) where the model should no longer appear as a black box. On the contrary, participants of a PM process should easily understand the relationship between the outputs of a simulation and the underlying conceptual model, and even contribute to modify it. For that purpose, we have developed an UML Class Diagram Editor that allows the participants to design the structure of their model. Then it generates the source code of the model that can be loaded on Cormas, an ABM platform dedicated to natural resources management. Conversely, the editor is also able to read existing code to generate a diagram by reverse engineering. As autonomous software, our editor can produce standard XMI files and other types of codes. Currently, Cormas is the targeted platform, but code generation can be easily available for other platforms. Obviously the generated code does not enable to run a simulation. It contains only the structure of the model: classes, attributes (with their default value) and associations. Methods to instantiate a simulation and activate the agents still need to be implemented. Nevertheless, Cormas offers many useful methods of its generic classes that can be reused by the specific agents of the modellers. As Cormas is oriented toward interactive simulation, this code generator will greatly facilitate the involvement of stakeholders in participating in the design of models to explore scenarios regarding their own socio-environmental systems.

Keywords: ABM; UML; Class diagram; Code generation; reverse engineering; Participatory Modelling

1 INTRODUCTION

Designing and implementing an agent-based model (ABM) is a hard and complex task that requires a wide skill set including design, programming and often deep understanding of the modelled domain. Therefore a team of specialists accomplishes such endeavour, where each focuses on a single part. Although successful in several areas of software development, such approach does not hold in Participatory Modelling (PM). In PM, each member should participate or at least understand each step of the development process.

Implementing a model however presents a very real challenge to PM. Whereas models and notations are tailored to specific problems and needs of a particular domain, for programming it is often the opposite. For a correct implementation, the programmer has to consider and work with a wide range of problems and restrictions of the target programming language, platforms and tools. The idea of programming as such, that is — the process of formalizing steps to be executed — is not foreign to the PM stakeholders when done on an abstract level (such as writing cooking recipe like process, or drawing a simple activity diagram). But performing the process by writing source code for a specific language takes on a whole new level of complexity that requires specialized skills and experience not only to write code, but even to understand it, is much challenging. Additionally the implementation itself may be time-consuming as it scales at least linearly with the size of the domain model, and therefore may not be appropriate to perform during PM sessions from purely practical perspective. This kind of burden cannot be placed on stakeholders of PM and has to be done by a professional (or done poorly by inexperienced developer). In either case it alienates the stakeholders and negatively impacts PM. It is in our interest to abstract, eliminate, and automate the need to write any code at all

during participatory modelling sessions with stakeholders. To improve the current situation we have developed an UML Class Diagram Editor for the Cormas platform.

2 USING GRAPHICAL EDITORS IN CORMAS

The Cormas platform has been used for 18 years as an artefact to foster learning about agent-based simulation for renewable resource management (Bousquet et al., 1998). Among the existing generic ABM platforms, Cormas occupies a tiny, yet lively, place oriented towards participatory modelling (Bommel et al., 2015). Thanks to regular training sessions and an electronic forum, a community of users has been gradually established that has enabled a sharing of ideas, practices and knowledge with a specific focus on PM (Le Page et al., 2012). One aspect of the training that is regularly well appreciated by the participants is the part on UML. The UML formalism allows the participants to sit back and reflect upon their model and the code. This course helps them to understand the objects' concepts and to formalize their model (Le Page & Bommel, 2005; Bommel & Müller, 2007).

UML transcends any programming language and any computer platform, and “encourages the modeller to spend more time on modelling than on writing code” (Bersini, 2012). The purpose of UML has always been on modelling and not programming, and we are fully agreed with this vision. Nevertheless, the software community is moving in the direction towards tools for automatic code generation. The participants of our courses are expecting such tools that would prevent them to code. For these reasons, we have oriented the development of Cormas towards the use of UML editors to generate a part of the computer code.

Several different notations and models are aimed at addressing specific tasks during participatory modelling. In particular, in Cormas, we mainly use two types of diagrams:

- * UML class diagrams to capture the structural information of the domain — this includes both the analytical, and implementation model,
- * UML activity diagrams to describe a particular behaviour of an agent.

An editor for activity diagrams is already available in Cormas. It allows users through a simple graphical interface, to create an executable behaviour for an agent without the need to write any code. This tool has been used in PM with technicians and bovine breeders in Uruguay (Bommel et al., 2014).

As noted in the introduction, UML class diagrams are also of great interest for PM. But translating the domain into code is time-consuming, error-prone, and complex. It is also one of the major hurdles of the PM approach. For this reason the developed UML class diagram editor described in this paper attempts to address this issue. Additionally by having a custom-made UML class editor that is closely tied to Cormas, we gain additional benefits such as closer collaboration and addressing special issues. Naturally, we do not restrict the user of this editor to Cormas and XMI support for models interchange is also provided, both of which will be demonstrated later.

3 THE NECESSARY ELEMENTS TO DESIGN A CLASS DIAGRAM

The first step for designing a new ABM consists in defining the structure of the model. To do so, the PM participants decide on the important entities in the domain and represent them on a UML class diagram. As a framework, Cormas recognized three primary types of entities: social, spatial, and passive. Social entities are active agents that interact with the environment and whose behaviour should be defined, modified, and simulated. Spatial entities are possible locations where social entities can be located and with which they can interact. Finally passive objects are other kinds of entities such as messages, land covers, strategies, etc. Each of these three entities are root categories for further specializing hierarchies.

3.1 Model classes and connection with Cormas classes

When designing a new model, the modeller has to create classes for his domain entities and map them to Cormas entities (mapping means that he has to decide from which Cormas entities his

classes should inherit). The editor offers two basic approaches to do so: the first one is to select a “stereotype” for a particular entity: a stereotype is a mechanism to extend the vocabulary of UML in order to create new elements. Graphically, it is rendered as a name enclosed by «guillemets». The Cormas meta-model therefore acts as a UML profile for the implemented model. The editor offers a hierarchical list of all the available Cormas entities the user can choose from. This approach makes the model visually more concise as it contains only elements pertinent to the domain. This however may be insufficient in some scenarios as it hides many relationships between the classes that are inherited from the core Cormas model. To answer this, a second approach is possible where the user can either display the classes from the core model to the diagram and then add generalization between the classes, or let the tool automatically expand the stereotypes to classes. That way both the model and the relevant portion of the core model can be viewed at once.

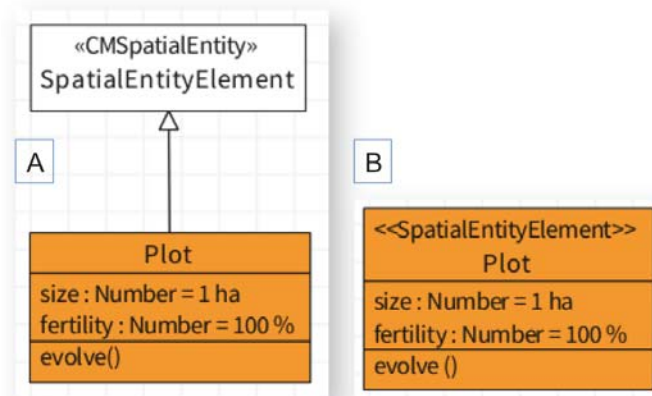


Fig. 1: Two options to represent the class “Plot” in Cormas-UML editor. *SpatialEntityElement*, a kind of spatial classes of Cormas framework, is displayed as super class of “Plot” (A), or the stereotype «SpatialEntityElement» is included into the “Plot” class (B).

Furthermore it could be achievable to also merge the core classes into stereotypes and show the inherited relationships directly on the subclasses whether they override it or not (see Fig. 2).

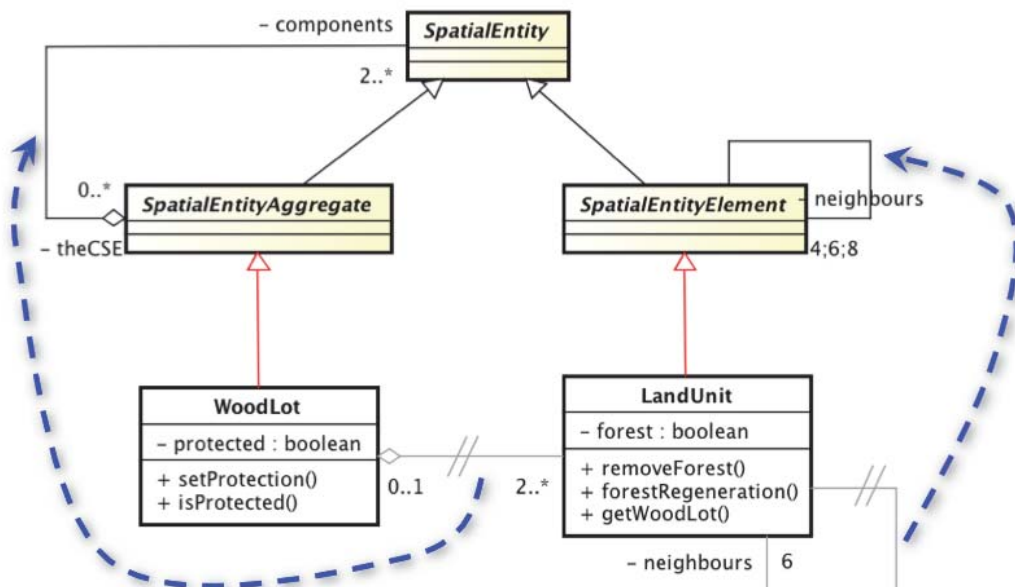


Fig. 2: Inherited associations. *WoodLot* and *LandUnit* are respectively two subclasses of *SpatialEntityAggregate* and *SpatialEntityElement* (from Cormas core). The ‘aggregate’ association between *WoodLot* and *LandUnit* presents a double bar: a stereotype that specifies that this association redefines another one (in this case, the recursive association between

SpatialEntityAggregate and *SpatialEntity*. It is also the case for the recursive “neighbours” association for *LandUnit* that redefines the one at level of *SpatialEntityElement*.

Overriding of relationships is possible in principle, as the closer one will simply take precedence, however visualization of that through association modifiers such as ‘union’ and ‘subsets’ is a subject of future work.

3.2 Attributes, values and units

Naturally, classes are not sufficient to design an ABM. The editor offers not just the option to add attributes (with name, multiplicities, and types), but also offers an extra interface to easily specify the default value from pre-defined categories. For attributes of Number type, the user can also define its unit that is displayed on the diagram. This option is not included in UML2.0 specifications, but it is useful for modellers and participants because it adds meaning to the model (see examples initial values and units in fig. 1: ‘ha’ and ‘%’). There is also the possibility to add some comments about an attribute, in order to explain its meaning or add bibliographic references on its value.

The default value will be used when a new agent or a spatial entity is created when initializing a new simulation, but also when a new entity is created during a simulation. Obviously, the value can be changed while the state of the entity is evolving. As public accessors are automatically defined, this attribute can be changed by another entity, or even by the user who is free to manipulate the attribute’s values for individual entities. The initial value manipulation interface was already available in Cormas, however it operated directly on the source code, as there was no built-in model editor. This has been addressed with the new editor as the new interface correctly operates on the model from which the values are generated into the source code.

3.3 The associations

Relationships represented by associations are an essential aspect of a structural model. In the current state, the editor offers the possibility to create standard bi-directional associations including association end multiplicities, navigabilities and aggregations. The following figure (Fig. 3) presents a basic “LandUse” model with an association between Plot and Cover.

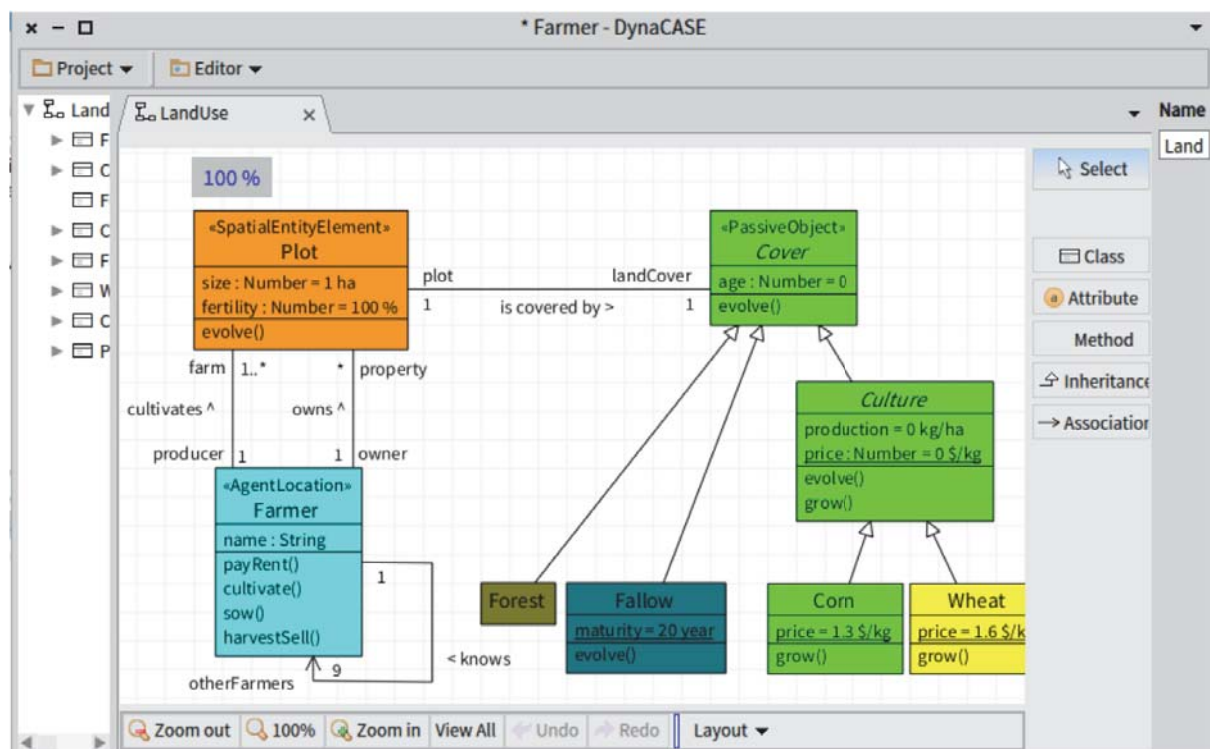


Fig 3: A complete view of the UML editor displaying a simple “LandUse” model.

This association, called “*is covered by*”, is a 1 - 1 association, which means that for each spatial plot of 1 ha is associated one instance of cover (called “*landCover*”). Four types of covers are available: Forest, Fallow, Corn or Wheat. Thus, the cover of a given plot can evolve (changing its age, its production, etc) and be replaced by another cover. For example, a fallow plot gets older every year, and when it reaches 20 years (age of maturity), it “becomes” a forest (ie. it is replaced on the plot by a new instance of Forest).

This model presents also two associations between Farmer and Plot. The first one, called “*owns*”, means that a farmer may own 0 to several plots. In that case, he plays the role of *owner* of these plots (which are his *properties*). The second one, called “*cultivates*”, specifies that a farmer has to cultivate at least one plot. These cultivated plots compose his farm. The interest of this double association is that it highlights the difference between owning one (or more) property and using a farm to cultivate it. This makes it possible for example to have a farmer agent owning 10 ha of land, but cultivating only one and renting the others to the other farmers (see example in fig. 4).

Finally, a recursive association, called “*knows*”, describes the fact that a farmer knows the *otherFarmers* (certainly for them to exchange and lease land).

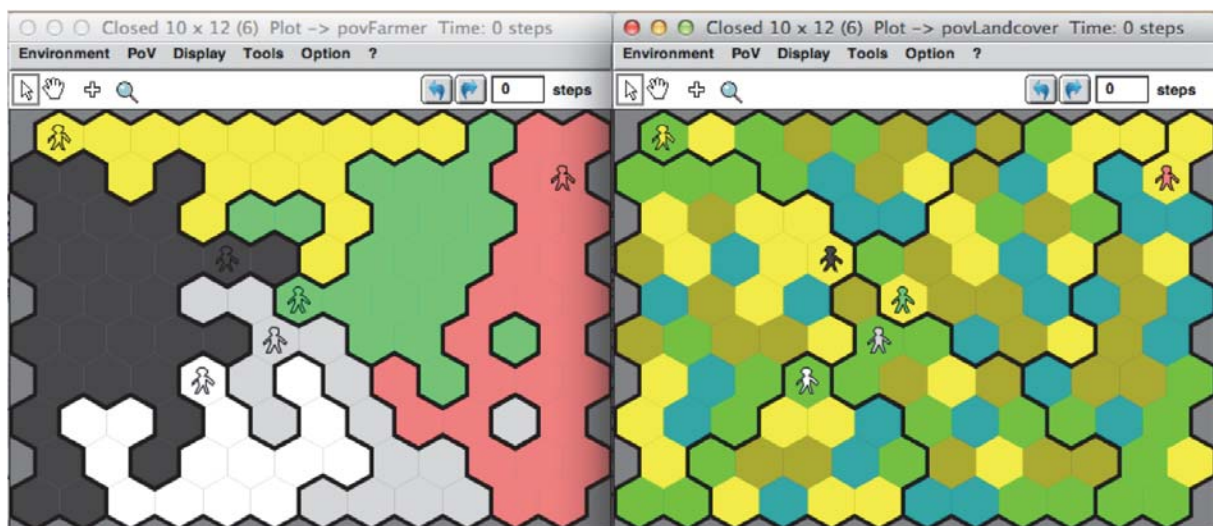


Fig 4: Two points of view of the Cormas spatial grid, showing 6 farmers and their farm (on the left), and (on the right) the same agents on their property; in that case, the land cover of each plot is displayed. After some exchanges of land for rental, the properties are not similar to the farms.

4 SOURCE-CODE GENERATION

Whilst having a modelling tool directly alongside Cormas, it is important from long-term perspective to continuously move toward complete model-driven development. Our current objective is on source code generation from the conceptual model, as the implementation, not the modelling, is a major hurdle for participants during PM.

4.1 Currently targeted languages

For the moment, the code generation targets two computer languages: Pharo-Smalltalk (<http://pharo.org>) to which Cormas is being ported and on which the UML class editor has been developed, and VisualWorks-Smalltalk (<http://www.cincomsmalltalk.com/main/products/visualworks/>) on which Cormas is currently implemented. The usual approach for model-driven development (MDD) is to use model as the primary source from which code should be generated for different platforms. In our case however the syntax between Pharo and VisualWorks (VW) is close enough that writing separate generators would be unnecessary. The main difference is the support of namespaces in VisualWorks, usually resolved in Pharo by prefixing class names with a prefix. Second difference is

about the libraries, however as we do not generate behaviour (code for method) this is not a concern for us.

Instead of writing two different generators, we apply a different strategy — we use the Smalltalk Interchange File Format (SIF) that is capable of representing smalltalk code in a dialect-neutral format and is supported by several different dialects, more importantly to us, SIF is supported by both Pharo and VisualWorks. This allows us to generate single source code for Pharo, export the code to SIF and import it into VisualWorks. The Cormas Editor generates VW code directly loadable into Cormas. Naturally as Pharo lacks namespaces, the imported classes will be namespace-less, however this is resolved by Cormas when loading the code of the model: from the model name, Cormas automatically creates a namespace then moves the model classes into this namespace. The process of import/export is bi-directional, therefore we can also copy existing VW code into Pharo, which will be important to reverse-engineer a model, as will be demonstrated later.

Generation of source code presents a significant challenge, as programming in general is a highly creative activity. However in constrained environment, the situation is quite different. This is indeed the case for Cormas models, as they have predefined constrained hierarchies (they inherit from limited number of Cormas classes) and likewise their size, expected relationships, and behaviours are limited by the capabilities of the simulator. For such constrained models, we can make uniform design decisions for the generated code.

The structural code is generated as follows: classes of the model (excluding those belonging to the core Cormas) are generated in their topological order (i.e. parent classes are generated before subclasses). Then the attributes are generated. Pharo and VW discern three types of attributes: instance variables private to each instance and available for all classes in the hierarchy (equivalent to protected attributes in e.g. Java), class variables (also known as shared variables) — shared between the class that defined it and all subclasses (equivalent to protected static attributes), and finally class instance variables that are private to each class (equivalent to private static attributes). The generator is not concerned with the last type as it is not used in Cormas models.

As the editor allow the users to define initial values for attributes, the generator permits two different approaches: (a) eager initialization, where the values are set in the constructor of the class (it's `#initialize` method), and (b) lazy initialization, where the value is set in the attribute's getter. Each approach has its limits, e.g. eager approach initializes attributes whether the values will be used or not which may consume a high amount of memory during simulation. Lazy initialization on the other hand forces the user to always access the attribute through the getter even from within the class itself, otherwise risking accessing undefined variables. For Cormas model generation purposes, we use the latter approach¹.

```
age
  <DCType: #Number multiplicity: #(1)>
  ^ age ifNil: [ age := 0 ]
```

Fig. 5: Example of code generation for the getter accessor of attribute 'age' with 0 as default value. Note that the first line represents a 'pragma': a special language used for annotations. This information is necessary for reverse-engineering (see next section).

4.2 Associations generation

UML associations are necessarily translated as attributes in the code. As showing by figure 6, the extremity role of an association could be also presented as attribute, but this information unnecessarily overloads the diagram: an association and its roles are more readable than an attribute and give more meaning to the model.

¹ A more technical description is available here: <https://github.com/dynacase/class-editor/blob/master/docs/code-generation/uml-to-code.md>

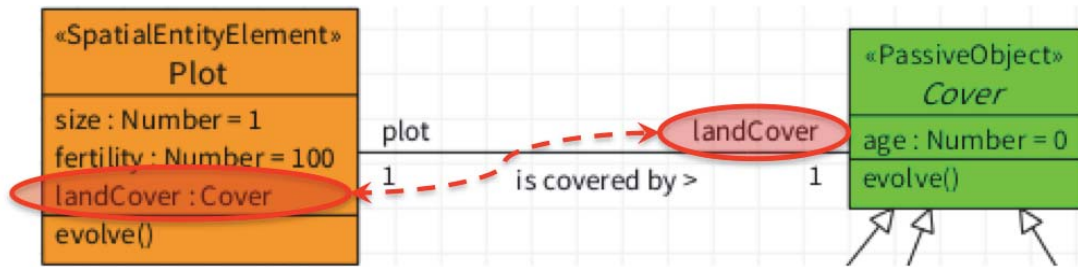


Fig. 6: Redundant information for “landCover” attribute

The most complex part of the code generation is the translation of associations into code. As it has been noted in the introduction, UML associations cannot be directly implemented. In ideal case, all association constraints should be enforced by the generated code, namely the navigability, multiplicity range, and bi-directional integrity. This presents a challenge as running programs have to operate outside of those constraints for a portion of time — a mandatory participation for both sides in bi-directional association cannot be enforced immediately as we cannot create both objects at once. Similarly updating opposite site of an association that is navigable only in the opposite direction may not be possible without exposing the non-navigable end. Different approaches attempt to address this issue in different languages to varying level of success, such as UML Java links e.g. by implementing associations as first-class objects placed between the objects in relationship. As we are operating on a constrained model, we have chosen for now a more direct approach: generating source code whose primary concern is maintaining (not enforcing!) the integrity of bi-directional relationships.

For each association, the generator considers whether the association is of one-directional type (nil -> 1 or nil -> many), or bi-directional type (1:1, 1:N, or N:M). Based on this information, it generates appropriate code. For nil->1 the code is equivalent to code for a simple attribute getter/setter as from UML perspective it is indeed equivalent to an attribute; for the other types, an appropriate code that updates the opposite side is generated. Thus if we set a different instance to an attribute, the setter code will first disconnect itself from the previous object by sending *nil* to the this object, and then it connects itself with the new one. The following code describes the setter accessor of the attribute *landCover* of *Plot*:

```
landCover: aCover
|oldRef|
landCover = aCover ifTrue: [ ^ self ].
oldRef := landCover.
landCover := aCover.
oldRef ifNotNil: [ oldRef forgetAttribute: #landCover ].
landCover ifNotNil: [ landCover plot: self ]
```

In this case, when setting a new instance of *Cover* (let say a *Corn*) to a plot in forest, the previous instance of *Forest* is disconnected from this plot (*oldRef forgetAttribute: #landCover*), then the corn instance is affected as *landCover* and the attribute ‘plot’ of this corn is connected to the plot. Care has to be taken whether a particular side is a single value or a collection, as appropriate ‘send’ message has to be used (adding to/removing from collections or using setters). Although this approach in principle violates the constraints of the UML model (namely non-navigable ends are sometimes visible or some associations do not have correct multiplicity at some points of the execution, in practice violating those constraints do not pose a problem as for purposes of our models they are either theoretical (navigability) or enforced by design of other parts of the system, such as spatial grid having a specific number of neighbours from principle.

5 REVERSE-ENGINEERING MODELS

One of interesting aspects of the editor is not just the ability to produce code from diagram, but also to generate and display the model’s diagram by reverse engineering from the model’s source code. This is essential if the system is not created by purely MDD means. Without MDD the code presents the primary artifact that is taken care of and which is maintained. Any external artifacts, such as documentation or the appropriate model have to be manually kept in sync, or discrepancies start to

occur, which may often lead to further separation of the code from the model, as model with already existing discrepancies is more likely to get ignored pushing it further into decay (syndrome of a broken window).

Reverse-engineering however presents a set of challenges of its own. As our current target languages (Pharo and VW) are dynamically typed, one of the essential components of the model — types and classes, cannot be determined from the source code alone. Methods exist to guess a type of an attribute or a method argument using analysis of the interactions the attribute/argument is in, however such approach will be by design imprecise, as significant power of those (Pharo/VW) systems stems precisely from late-binding and message dispatch — showing the actually used types only during the runtime. More importantly much information is simply not available or not decidable such as multiplicities and to large extent participation in associations. To address this problem, for new models we have decided to include this information in the generated code. To do so, we make use of so-called “pragmas” — a special language constructs that are used for annotations. This way, during the code generation, methods are annotated with ‘pragmas’ containing information necessary for full model reverse-engineering. Naturally this does not ensure that a discrepancy does not occur, as a programmer is free to modify both the source code and the pragmas, however it significantly improves the situation. Likewise by including this information alongside the code, we raise the probability that a programmer will update pragma next to the code he is changing, rather than when he has to switch to a different tool or environment to update the matching model. In ideal MDD scenario, only model should ever be updated, however as we are currently do not have the appropriate toolset, and we have many existing models, having an option to reverse-engineer a model improves our standing.

CONCLUSION

The UML editor allows participants of a PM experiment to design their model then to generate the structure of the source-code. This code can then be loaded into the Cormas platform. Nevertheless, to simulate this model, programming skills in Smalltalk are still needed to code the methods. Cormas offers an activity diagram editor to design the main behaviour of the agents, but the basic operations to be used as elements of this diagram, should be coded. However, both UML editors prevent the modeller much of computer coding.

Even if the class diagram editor presented here is independent from any platform, a specific version that focuses on Cormas is available. As the current version of Cormas runs on VisualWorks, the generated code is targeted to this OO language. Furthermore, some additional options to standard UML are provided in order to help the participants of a PM process to give more meaning to their model: colored classes (that are translated in the Cormas simulator as the default colour of the instances), initial value and unit of attributes, inheritance from entity classes of the Cormas framework. In the same idea, the UML formalism has been simplified: the visibility of attributes and methods is not displayed because these notations (+, -, #, ~) disturb the participants and are not essential. Only binary associations are available (association-classe, n-ary association, or qualified association are not proposed).

The UML editor is currently in testing phase and will be soon available on the website of Cormas. We hope it will allow young scientists without computer skills to engage in ABM. We believe it will also be useful for PM with local stakeholders to create their own model. Then, beyond participation in the analysis of simulations, one could talk about true participation in model design.

REFERENCES

- Bersini, H. (2012). UML for ABM. *Journal of Artificial Societies and Social Simulation*. <<http://jasss.soc.surrey.ac.uk/15/1/9.html>>
- Bommel, P., Dieguez, F., Bartaburu, D., Duarte, E., Montes, E., Pereira Machin, M., ... Morales Grosskopf, H. (2014). A Further Step Towards Participatory Modelling. Fostering Stakeholder Involvement in Designing Models by Using Executable UML. *Journal of Artificial Societies and Social Simulation*, 17(1), 6. <<http://jasss.soc.surrey.ac.uk/17/1/6.html>>

- Bommel, P., & Müller, J. P. (2007). An Introduction to UML for Modelling in the Human and Social Science", in Phan, Amblard (Eds), *Agent-based Modelling and Simulation in the Social and Human Sciences*. The Bardwell Press, p. 273-294. Oxford.
- Bousquet, F., Bakam, I., Proton, H., & Page, C. L. (1998). Cormas: Common-pool resources and multi-agent systems. In A. P. del Pobil, J. Mira, & M. Ali (Éd.), *Tasks and Methods in Applied Artificial Intelligence* (p. 826-837). Springer Berlin Heidelberg.
<http://link.springer.com/chapter/10.1007/3-540-64574-8_469>
- Le Page, C., Becu, N., Bommel, P., & Bousquet, F. (2012). Participatory Agent-Based Simulation for Renewable Resource Management: The Role of the Cormas Simulation Platform to Nurture a Community of Practice. *Journal of Artificial Societies & Social Simulation*, 15(1).
<<http://jasss.soc.surrey.ac.uk/15/1/10.html>>
- Le Page, C., & Bommel, P. (2005). A methodology for building agent-based simulations of common-pool resources management: from a conceptual model designed with UML to its implementation in CORMAS. *Companion Modeling and Multi-Agent Systems for Integrated Natural Resource Management in Asia*, 327–349.